**DockerFile的概念**

|  |
| --- |
| Docker通过读取Dockerfile里面的内容可以自动build image，Dockerfile是一个包含了build过程中需要执行的所有命令的文本文件。也可以理解为Dockfile是一种被Docker程序解释的脚本，由一条一条的指令组成，每条指令对应Linux系统下面的一条命令，由Docker程序将这些Dockerfile指令翻译成真正的Linux命令。Dockerfile有自己书写格式和支持的命令，Docker程序解决这些命令间的依赖关系，  Dockerfile 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。有了 Dockerfile，当我们需要定制自己额外的需求时，只需在 Dockerfile 上添加或者修改指令，重新生成 image 即可，省去了敲命令的麻烦。 |

**DockerFile指令参数**

|  |
| --- |
| Dockerfile 分为四部分：**基础镜像信息、维护者信息、镜像操作指令、容器启动执行指令**。一开始必须要指明所基于的镜像名称，接下来一般会说明维护者信息；后面则是镜像操作指令，例如 RUN 指令。每执行一条RUN 指令，镜像添加新的一层，并提交；最后是 CMD 指令，来指明运行容器时的操作命令。 |
| 1、第一行必须指定 基础镜像信息FROM ubuntu #  2、维护者信息MAINTAINER docker\_user docker\_user@email.com #  3、镜像操作指令RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.listRUN apt-get update && apt-get install -y nginxRUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf #  4、容器启动执行指令CMD /usr/sbin/nginx |

**FROM指令详解:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| : 所谓定制镜像，那么就一定是以一个镜像为基础，在其上进行修改定制。就像我们之前运行了一个Nginx的容器，在其上面修改一样，基础容器是必需指定的。而FROM就是指定基础镜像，因此在DockerFile中，FROM是必备指定，并且必需是第一条指令！  docker基础镜像分类:  镜像官网  操作系统基础镜像  busybox  Alpine  CentOS  Ubuntu  Debian  编程语言基础镜像  Java基础镜像  Python基础镜像  NodeJs基础镜像  应用基础镜像  Nginx基础镜像  Tomcat基础镜像  Jetty基础镜像  其它基础镜像例子  Maven基础镜像  Jenkins基础镜像  GitLab基础镜像 操作系统基础镜像 比如你要从Linux操作系统基础镜像开始构建，可以参考下表来选择合适的基础镜像：   |  |  |  | | --- | --- | --- | | 镜像名称 | 大小 | 使用场景 | | [busybox](https://hub.docker.com/_/busybox/) | 1.15MB | 临时测试用 | | [alpine](https://hub.docker.com/_/alpine/) | 4.41MB | 主要用于测试，也可用于生产环境 | | [centos](https://hub.docker.com/_/centos/) | 200MB | 主要用于生产环境，支持CentOS/Red Hat，常用于追求稳定性的企业应用 | | [ubuntu](https://hub.docker.com/_/ubuntu/) | 81.1MB | 主要用于生产环境，常用于人工智能计算和企业应用 | | [debian](https://hub.docker.com/_/debian/) | 101MB | 主要用于生产环境 |   **scratch** 空镜像 busybox 可以将busybox理解为一个超级简化版嵌入式Linux系统。  **Alpine**  Alpine是一个面向安全的、轻量级的Linux系统，基于musl libc和busybox。  FROM scratch  ADD rootfs.tar.xz /  CMD ["/bin/sh"]  **Debian**  另一个非常出色的Linux发行版  FROM scratch  ADD rootfs.tar.xz /  CMD ["bash"]  **centos/fedora/ubuntu**  可以理解CentOS是RedHat的社区版  FROM scratch  ADD centos-7.4.1708-docker.tar.xz /  LABEL name="CentOS Base Image" \  vendor="CentOS" \  license="GPLv2" \  build-date="20170911"  CMD ["/bin/bash"]  docker是容器不同于虚拟机，运行之后只有应用程序在里面运行，**基础镜像只是为应用程序提供依赖包**。应用程序是主，基础镜像是次。  如果没有必要，选择最小的依赖环境alpine，然后在这个基础上制作自己的镜像。 |

## RUN

RUN指令是用来执行命令行命令的，由于命令行的强大功能，RUN指令是定制镜像时最常用的指令之一。其格式有两种：

**shell格式**：就像在命令行中输入的Shell脚本命令一样，比如之前的：

echo '<h1>Hello Docker!</h1>' > /usr/share/nginx/html/index.html

**exec格式**：像是函数调用的格式，例如：

apt-get update

mkdir -p /usr/src/redis

DockerFile的每一个指令都会新构建一层，RUN命令也不例外。每一个RUN行为，都会新建立一层，然后在其上执行命令，执行完毕后，提交这一层的修改，构成新的镜像！

## COPY

COPY指令将从上下文目录中的指定路径下的文件或文件夹复制到新的一层的镜像内的指定路径之下，格式为：

COPY <源路径> ... <目标路径>

COPY ./test1.py ./test2.py /test/

COPY ./t\*.py /test/

COPY ./test?.py /test/

## ADD

ADD指令和COPY的格式和性质基本一致，只不过是在COPY的基础上增加了一些功能。例如ADD指定中，源路径可以是一个远程URL，Docker引擎会自动帮我们将远程URL的文件下载下来到目标路径下，例如：

ADD http://192.168.0.89:5000/test.py /test/

其实ADD命令并不实用，并不推荐使用！！！

## CMD

CMD指令与RUN指令相似，也具有两种格式：

* **shell格式**：**CMD <命令>**
* **exec格式**：**CMD [“可执行文件”, “参数1”, “参数2”, …]**

CMD就是指定默认的容器主进程的启动命令的。

## ENTRYPOINT

ENTRYPOINT指令和CMD指令目的一样，都是指定容器运行程序及参数，并且与CMD一样拥有两种格式的写法：

* **shell格式**：**ENTRYPOINT <命令>**
* **exec格式**：**ENTRYPOINT [“可执行文件”, “参数1”, “参数2”, …]**

## ENV

ENV指令用于设置环境变量，格式有两种：

ENV

ENV = = …

这个指令非常简单，就是用于设置环境变量而已，无论是接下来的指令，还是在容器中运行的程序，都可以使用这里定义的环境变量。例如：

FROM ubuntu:16.04

ENV MODE=test

RUN apt-get update && apt-get install -y curl && curl http://192.168.0.89:5000/$MODE && rm -rf /var/lib/apt/lists/\*

如果你要设置多个环境变量，为了美观，你可以使用\来进行换行。多个环境变量的隔开，使用空格进行隔开的，如果某个环境变量的值是由一组英文单词构成，那么你可以将其使用""进行圈起来。如下：

FROM ubuntu:16.04

RUN MODE=test DESCRITPION="ios 12" \

TITLE="iphone"

接下来，将这个Dockerfile构建成镜像，然后以此镜像为基础创建并启动一个容器，在容器中，我们调用这个环境变量，仍然是有用的！！！

值得注意的是，如果你想通过CMD或者ENTRYPOINT指令的exec格式来打印环境，就像下面这样：

CMD ["echo", $MODE]

CMD ["echo", "$MODE"]

这样都是不能正确输出环境变量的值的，你可以改成exec格式来执行shell命令，如下：

CMD ["sh", "-c", "echo $MODE"]

如此，就能正确输出环境变量的值了！

## ARG

构建参数ARG和ENV指令一样，都是设置环境变量。与之不同的是，ARG设置的环境变量只是在镜像构建时所设置的，在将来容器运行时是不会存在这些环境变量的。但是不要因此就用ARG来保存密码之类的信息，因为通过docker history还是能够看得到的。ARG指令与ENV指令的使用类似，如下：

FROM ubuntu:16.04

ARG app="python-pip"

RUN apt-get update && apt-get install -y $app && rm -rf /var/lib/apt/lists/\*

ARG构建参数可以通过docker run命令中的--build-arg参数来进行覆盖

VOLUME

VOLUME指令用于构建镜像时定义匿名卷，其格式有两种：

VOLUME <路径>

VOLUME [“<路径1>”, “<路径2>”, …]

之前我们说过，容器存储层应该保持无状态化，容器运行时应尽量保持容器内不发生任何写入操作，对于需要保存动态数据的应用，其数据文件应该将其保存在数据卷中(VOLUME)

定义一个匿名卷：

FROM ubuntu:16.04

VOLUME /data

定义多个匿名卷：

FROM ubuntu:16.04

VOLUME ["/data", "/command"]

这里的/data和/command目录在容器运行时会自动挂载为匿名卷，任何向/data和/command目录中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化！容器匿名卷目录指定可以通过docker run命令中指定-v参数来进行覆盖

## EXPOSE

EXPOSE指令是声明运行时容器服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在Dockerfile中这样声明有两个好处：一个是帮助镜像使用者更好的理解这个镜像服务的守护端口，另一个作用则是在运行时使用随机端口映射时，也就是docker run -p命令时，会自动随机映射EXPOSE端口。

要将EXPOSE和在运行时使用-p <宿主>:<容器端口>区分开来，-p是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而EXPOSE仅仅是声明端口使用什么端口而已，并不会自动在宿主进行端口映射。

WORKDIR

使用WORKDIR指令来制定工作目录(或者称为当前目录)，以后各层操作的当前目录就是为指定的目录，如果该目录不存在，WORKDIR会自动帮你创建目录，如下：

FROM ubuntu:16.04

WORKDIR /data/test

RUN mkdir docker && echo "test" > demo.txt

当我们使用docker build构建此镜像，并使用docker run命令进行创建和启动容器之后，会发现目录被自动切换到了/data/test，并且在当前目录下有一个文件夹docker，在docker下有一个文件domo.txt并且有相应的内容。我们还可以为特定的指令指定不同的工作目录，如下：

FROM ubuntu:16.04

WORKDIR /data/test

RUN mkdir docker

WORKDIR /data/test/docker

RUN echo "test" > demo.txt

这样，Dockerfile中两次RUN指令的操作都在不同的目录下进行，最终容器会切换到最后一次WORKDIR指令下的目录。

WORKDIR指令可以通过docker run命令中的-w参数来进行覆盖

USER

USER指令用于将会用以什么样的用户去运行，例如：

FROM ubuntu:16.04

USER docker

基于该镜像启动的容器会以docker用户的身份来运行，我们可以指定用户名或者UID，组名或者GID，或者两者的结合，如下：

FROM ubuntu:16.04

USER user

USER user:group

USER uid

USER uid:gid

USER user:gid

USER uid:group

USER指令可以在docker run命令中的-u参数进行覆盖

## HEALTHCHECK

HEALTHECHECK指令是告诉Docker该如何判断容器的状态是否正常，这是1.12引入的新指令，其格式有两种：

HEALTHCHECK [options] CMD <命令>：检查容器健康状态的命令

HEALTHCHECK NONE：如果基础镜像有健康检查指令，这一行将会屏蔽掉其健康检查指令

HEALTHECHECK支持下列选项：

–interval=<间隔>：两次检查的时间间隔，默认为30s

–timeout=<时长>：健康检查命令运行超时时间，如果超过这个时间，本次健康检查将会判定为失败，默认为30s

–retries=<次数>：当连续失败指定次数之后，则将容器状态视为unhealthy，默认为3次

在没有HEALTHCHECK指令之前，Docker引擎只可以通过容器内主进程是否退出来判断容器状态是否异常。很多情况下这没有问题，但是如果程序进入了死锁状态，或者死循环状态，应用进程并不退出，但是该容器已经无法继续提供服务了。在1.12之前，Docker引擎不会检测到容器的这种状态，从而不会重新调度，导致可能容器已经无法提供服务了却仍然还在接收用户的请求。

假设我们有个镜像是最简单的Web服务，我们希望增加健康检查来判断Web服务是否在正常工作，我们可以用curl来帮助判断，其Dockerfile的HEALTHCHECK可以这么写：

FROM nginx

RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/\*

HEALTHCHECK --interval=5s --timeout=3s CMD curl -fs http://localhost/ || exit 1

接下来，我们将该Dockerfile编译构建成一个镜像，并以此镜像为基础创建并启动一个容器。此时，我们使用docker container ls命令来查看容器的状态，如下：

root@ubuntu:~/docker# docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

036b91eea00d nginx:v1.2 "nginx -g 'daemon of…" 7 seconds ago Up 6 seconds (healthy) 0.0.0.0:80->80/tcp web

我们再STATUS这一列中可以看到，状态未healthy。如果我们快速的多次执行docker container ls的话，会发现STATUS状态是由health: starting最后变为healthy，当然如果容器未在正常工作，最后的状态将会变为unhealthy

这里，我们设置了每5s检查一次，如果检查时间超过3s没有响应就视为失败。||符号左边的命令执行结果为假，右边的命令才会执行！

为了帮助排除故障，健康检查命令的输出会被存储于健康状态里，我们可以使用docker inspect命令来进行查看：

root@ubuntu:~/docker# docker inspect --format '{{json .State.Health}}' web | python3 -m json.tool

{

"Status": "healthy",

"FailingStreak": 0,

"Log": [

{

"Start": "2018-07-17T21:15:05.900643297+08:00",

"End": "2018-07-17T21:15:05.968989028+08:00",

"ExitCode": 0,

"Output": "<!DOCTYPE html>\n<html>\n<head>\n<title>Welcome to nginx!</title>\n<style>\n body {\n width: 35em;\n margin: 0 auto;\n font-family: Tahoma, Verdana, Arial, sans-serif;\n }\n</style>\n</head>\n<body>\n<h1>Welcome to nginx!</h1>\n<p>If you see this page, the nginx web server is successfully installed and\nworking. Further configuration is required.</p>\n\n<p>For online documentation and support please refer to\n<a href=\"http://nginx.org/\">nginx.org</a>.<br/>\nCommercial support is available at\n<a href=\"http://nginx.com/\">nginx.com</a>.</p>\n\n<p><em>Thank you for using nginx.</em></p>\n</body>\n</html>\n"

}

]

}

和CMD、NETRYPOINT一样，HEALTHCHECK指令只可以出现一次，如果有多个HEALTHCHECK指令，那么只有最后一个才会生效！！！

## ONBUILD

ONBUILD是一个特殊的指令，它后面跟着的是其他指令，比如COPY、RUN等，而这些命令在当前镜像被构建时，并不会被执行。只有以当前镜像为基础镜像去构建下一级镜像时，才会被执行。格式为：ONBUILD <其他指令>

Dockerfile中的其他指令都是为了构建当前镜像准备的，只有ONBUILD指令是为了帮助别人定制而准备的。例如：

from ubuntu:16.04

WORKDIR /data

ONBUILD RUN mkdir test

此时，我们以此Dockerfile进行构建镜像ubuntu:test，并以此镜像为基础创建并启动一个容器，进入容器后，容器会自动切换到WORKDIR指令下的目录，此时我们使用ls命令会发现在工作目录下，并未创建test文件夹，如下：

root@ubuntu:~/docker# docker run -it ubuntu:test

root@3a8f912fd23b:/data# ls

root@3a8f912fd23b:/data#

此时，我们再创建一个Dockerfile，只需一个FROM指令即可，使其继承刚刚我们构建的ubuntu:test镜像，如下：

FROM ubuntu:test

我们再以此Dockerfile构建镜像ubuntu:test\_onbuild，并以此镜像为基础创建并启动一个容器，进入容器后，容器会自动切换到WORKDIR指令下的目录，此时我们使用ls命令会发现在工作目录下，已经创建好了一个名为test的文件夹，如下：

root@ubuntu:~/docker# docker run -it ubuntu:test\_onbuild

root@5394e605b6ea:/data# ls

test

## LABEL

LABEL指令可以为镜像指定标签，其格式为：LABEL <key1>=<value1> <key2>=<value2> ...

LABEL后面是键值对，多个键值对以空格进行隔开，如果value中包含空格，请使用""将value进行圈起来，如下：

FROM ubuntu:16.04

LABEL name=test

LABEL description="a container is used to test"

我们知道，DockerFile的每一个指令都会新构建一层，所以，上面的LABEL我们可以写成一条指令，用空格进行隔开，如下：

FROM ubuntu:16.04

LABEL name=test description="a container is used to test"

为了美观，我们还可以使用\符号进行换行操作。

要查看镜像的标签，我们可以使用docker inspect命令，如下：

root@ubuntu:~# docker inspect --format '{{json .Config.Labels}}' test | python3 -m json.tool

{

"description": "a container is used to test",

"name": "test"

}

其中“test”为容器名称！

值得注意的是，这里的标签并非是我们一开始将镜像名称中的<仓库>:<标签>，这两者是不一样的！这里标签，类似于签条，注解之类的意思

MAINTAINER

MAINTAINER指令用于指定生成镜像的作者名称，其格式为：MAINTAINER <name>

MAINTAINER指令已经被弃用，可以使用LABEL指令进行替代，如下：

LABEL maintainer='Stephen Chow'

**DockerFile在线安装Mysql**

**编写 DockerFile文件 开头必须大写**

|  |
| --- |
| FROM centos:6.7  MAINTAINER chenyufeng "yufengcode@gmail.com"  # 使用yum的方式安装mysql；  RUN yum install -y mysql-server mysql  # 安装完成以后，执行以下命令。配置用户名密码相关信息；  RUN /etc/init.d/mysqld start &&\  mysql -e "grant all privileges on \*.\* to 'root'@'%' identified by '123456' WITH GRANT OPTION ;"&&\  mysql -e "grant all privileges on \*.\* to 'root'@'localhost' identified by '123456' WITH GRANT OPTION ;"&&\  mysql -u root -p123456 -e "show databases;"  # 镜像暴露3306端口；  EXPOSE 3306  # 容器启动后执行以下命令，启动mysql；  CMD ["/usr/bin/mysqld\_safe"] |

**执行DockerFile**

**docker build -t mysql:5.5 .**

**后面的.标识当前目录**

## 执行完成后、查看安装镜像:

|  |
| --- |
| [root@localhost mysql]# docker images  REPOSITORY TAG IMAGE ID CREATED SIZE  centos mysql 80aff47279d9 About an hour ago 393MB  centos 6.7 9f1de3c6ad53 3 months ago 191MB |

启动镜像:

|  |
| --- |
| sudo docker run --name=mysqlserver -d -P centos:mysql |

查看进程:

|  |
| --- |
| [root@localhost mysql]# docker ps  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  55a3bb646113 centos:mysql "/usr/bin/mysqld\_safe" 18 seconds ago Up 17 seconds 0.0.0.0:32770->3306/tcp mysqlserver |

**DockerFile安装SpringBoot应用**

**执行dockerFile**

|  |
| --- |
| docker build -t admin:0.0.1 . (后面的.代表当前目录) |

**联通其他应用**

|  |
| --- |
| docker run -d -p 8081:8080 --link mysql01:mysqldb -t admin:0.0.1  --link mysqldb:mysqldb 注意：--link参数格式为--link name:alias,其中name是要链接的容器名称，alias是这个连接的名称。 |